```python
import collections


class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Codec:
    def serialize(self, root):
        """Encodes a tree to a single string using level-order traversal (BFS)."""
        if not root:
            return "null"

        queue = collections.deque([root])
        result = []

        while queue:
            node = queue.popleft()
            if node:
                result.append(str(node.val))
                queue.append(node.left)
                queue.append(node.right)
            else:
                result.append("null")  # Mark null nodes

        return ",".join(result)

    def deserialize(self, data):
```

```python
        """Decodes the string back into a binary tree using BFS."""
        if data == "null":
            return None

        nodes = data.split(",")
        root = TreeNode(int(nodes[0]))
        queue = collections.deque([root])
        index = 1

        while queue:
            node = queue.popleft()
            if nodes[index] != "null":
                node.left = TreeNode(int(nodes[index]))
                queue.append(node.left)
            index += 1

            if nodes[index] != "null":
                node.right = TreeNode(int(nodes[index]))
                queue.append(node.right)
            index += 1

        return root

# Example Usage
codec = Codec()
root = TreeNode(1, TreeNode(2), TreeNode(3, TreeNode(4), TreeNode(5)))

serialized = codec.serialize(root)
print("Serialized:", serialized)
```

```python
deserialized = codec.deserialize(serialized)
print("Deserialized Root Value:", deserialized.val)




from collections import deque

class MaxFlow:
    def __init__(self, graph):
        self.graph = [row[:] for row in graph]  # Copy graph to preserve original
        self.V = len(graph)  # Number of vertices

    def bfs(self, source, sink, parent):
        """Finds an augmenting path using BFS and fills parent array."""
        visited = [False] * self.V
        queue = deque([source])
        visited[source] = True

        while queue:
            u = queue.popleft()
            for v in range(self.V):
                if not visited[v] and self.graph[u][v] > 0:  # If capacity exists
                    queue.append(v)
                    visited[v] = True
                    parent[v] = u  # Store path

                    if v == sink:  # If sink is reached, return True
```

```python
            return True
    return False

def ford_fulkerson(self, source, sink):
    """Returns the maximum flow from source to sink."""
    parent = [-1] * self.V
    max_flow = 0

    while self.bfs(source, sink, parent):  # While augmenting path exists
        path_flow = float('Inf')
        v = sink

        # Find the bottleneck (minimum capacity in path)
        while v != source:
            u = parent[v]
            path_flow = min(path_flow, self.graph[u][v])
            v = u

        # Update residual capacities
        v = sink
        while v != source:
            u = parent[v]
            self.graph[u][v] -= path_flow
            self.graph[v][u] += path_flow  # Reverse flow
            v = u

        max_flow += path_flow  # Add to total flow

    return max_flow
```

```python
# Example Usage
graph = [[0, 16, 13, 0, 0, 0],
         [0, 0, 10, 12, 0, 0],
         [0, 4, 0, 0, 14, 0],
         [0, 0, 9, 0, 0, 20],
         [0, 0, 0, 7, 0, 4],
         [0, 0, 0, 0, 0, 0]]

max_flow_solver = MaxFlow(graph)
source, sink = 0, 5
print("Maximum Flow:", max_flow_solver.ford_fulkerson(source, sink))
```

```python
def min_edit_distance(word1, word2):
    m, n = len(word1), len(word2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize base cases
    for i in range(m + 1):
        dp[i][0] = i  # Deleting all characters from word1
    for j in range(n + 1):
        dp[0][j] = j  # Inserting all characters to word1

    # Fill DP table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]:  # No change needed
```

```
            dp[i][j] = dp[i - 1][j - 1]
          else:
            dp[i][j] = min(
              dp[i - 1][j] + 1,    # Delete
              dp[i][j - 1] + 1,    # Insert
              dp[i - 1][j - 1] + 1 # Replace
            )

  return dp[m][n]

# Example Usage
word1 = "kitten"
word2 = "sitting"
print("Edit Distance:", min_edit_distance(word1, word2))
```

```
class TreeNode:
  def __init__(self, val=0, left=None, right=None):
    self.val = val
    self.left = left
    self.right = right

def inorder_traversal(root, elements):
  """Helper function to perform in-order traversal."""
  if root:
    inorder_traversal(root.left, elements)
```

```python
        elements.append(root.val)
        inorder_traversal(root.right, elements)

def kth_smallest_element(root, k):
    elements = []
    inorder_traversal(root, elements)
    return elements[k - 1]  # k-th smallest (1-based index)

# Example Usage
root = TreeNode(3)
root.left = TreeNode(1)
root.left.right = TreeNode(2)
root.right = TreeNode(4)

k = 2
print("K-th Smallest Element:", kth_smallest_element(root, k))
```

```python
def max_product_subarray(nums):
    if not nums:
        return 0  # Edge case: Empty list

    max_product = min_product = result = nums[0]

    for i in range(1, len(nums)):
```

```
        if nums[i] < 0:  # Swap max & min when encountering a negative number
            max_product, min_product = min_product, max_product

        max_product = max(nums[i], max_product * nums[i])
        min_product = min(nums[i], min_product * nums[i])

        result = max(result, max_product)  # Update global max

    return result

# Example Usage
nums = [2, 3, -2, 4]
print("Maximum Product Subarray:", max_product_subarray(nums))
```

```
from collections import defaultdict

def find_all_paths(graph, start, end, path=[], all_paths=[]):
    path.append(start)  # Add current node to path

    if start == end:
        all_paths.append(list(path))  # Store a copy of the valid path
    else:
        for neighbor in graph[start]:  # Explore neighbors
            if neighbor not in path:  # Avoid cycles
                find_all_paths(graph, neighbor, end, path, all_paths)
```

```python
    path.pop()  # Backtrack to explore other paths
    return all_paths

# Example Usage
graph = {
    0: [1, 2],
    1: [3],
    2: [3, 4],
    3: [5],
    4: [5],
    5: []
}

start_node, end_node = 0, 5
all_paths = find_all_paths(graph, start_node, end_node)
print("All Paths:", all_paths)
```